Dynamic Space-Time Scheduling for GPU Inference

Paras Jain, Xiangxi Mo,
Ajay Jain[§], Harikaran Subbaraj, Rehan Sohail Durrani,
Alexey Tumanov, Joseph Gonzalez, Ion Stoica
University of California, Berkeley
[§]Massachusetts Institute of Technology
{paras_jain, xmo, hsubbaraj, rdurrani, atumanov,
jegonzal, istoica}@berkeley.edu, ajayjain@mit.edu

Abstract

Serving deep neural networks in latency critical interactive settings often requires GPU acceleration. However, the small batch sizes typical in online inference results in poor GPU utilization, a potential performance gap which GPU resource sharing can address. In this paper, we explore several techniques to leverage both temporal and spatial multiplexing to improve GPU utilization for deep learning inference workloads. We evaluate the performance trade-offs of each approach with respect to resource-efficiency, latency predictability, and isolation when compared with conventional batched inference. Our experimental analysis suggests at least a 5x potential for improved utilization through the exploration of more advanced spatial and temporal multiplexing strategies. Our preliminary prototype of a dynamic space-time scheduler demonstrates a 3.18x speedup over space-only multiplexing and a 7.76x speedup over time-only multiplexing, while also providing better isolation and latency predictability.

1 Introduction

GPUs are essential to deep learning. By leveraging substantial parallelism, high memory bandwidth, and tensor acceleration, GPUs are able to quickly compute activations over large batches of inputs. NVIDIA's datacenter-class V100 GPU, for example, packs more than 120 TFLOP/s of half-precision matrix multiply-and-accumulate performance designed specifically for deep learning workloads. As deep learning is deployed in applications ranging from video monitoring, language translation, and speech recognition, there is an emerging need for parallel hardware accelerators to support inference. While there are numerous specialized inference processors, the widespread availability of GPUs and their support for general deep learning models renders GPUs indispensable for inference.

DNN training is computationally expensive but relatively infrequent, while online inference needs to scale to billions of queries per day and is rapidly outpacing training in datacenters [11]. Key metrics for revenue-critical applications can dramatically suffer with an increased application latency [23]. In spite of tight end-to-end latency budgets (< 100ms), we note an alarming trend that inference latency on a CPU has been on a rise (Figure 1). The researcher's appetite for better accuracy leads to ever-increasing model size and complexity; as an example, the state-of-the-art SENet-184 [14] model has a 4.1s CPU inference latency. Given that increases in interactive query latencies leads to losses in revenue, CPUs cannot support today's interactive model serving workloads which leaves GPUs an obvious favorite.

While training workloads continue to scale and can often easily saturate modern GPUs, ML inference has distinctly different performance requirements that often result in poor GPU utilization. In contrast to throughput-oriented model training, revenue-critical inference workloads must meet latency objectives with queries often arriving stochastically. In practice, online inference queries

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

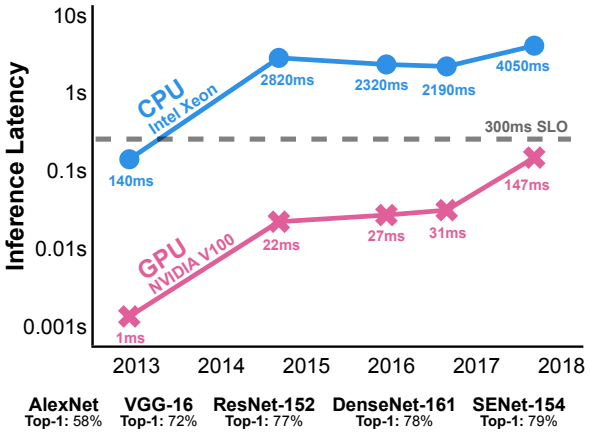

Figure 1: DNN model complexity is increasing over time on CPUs and GPUs. Most models fail to meet the 300ms latency SLO on a CPU. Models from left-to-right: AlexNet [17], VGG-16 [24], ResNet-152 [13], DenseNet-161 [15] and SENet-184 [14].

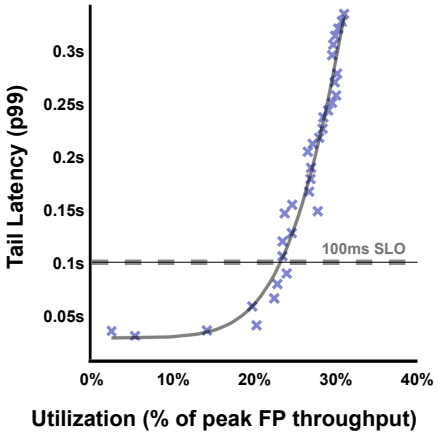

Figure 2: In order to meet latency SLOs, small batch sizes must be used resulting in low GPU utilization. The largest batch size for ResNet-50 (plotted) within the SLO is 26, but only achieves 28% of peak V100 floating-point throughput.

often cannot realize the high levels of parallelism that offline iterative minibatch training achieves; lower parallelism leads to poor GPU utilization in practice.

Small batch sizes are an unfortunate reality for online inference with SLOs (Figure 2), which leads to low utilization. The issue raised by small inference batch sizes is exacerbated as model complexity grows over time, pushing GPU inference latencies to approach interactive SLOs from below (as noted in Figure 1). Given that inference workloads must run continuously and respond to highly variable demand, capacity must be provisioned for demand peaks which lowers GPU utilization even further.

The current practice of exclusive access to a GPU cannot scale due to the low utilization on current hardware. We notice that small batch sizes can lead the GPU to low utilization under 15%. This problem affects other throughput-oriented accelerators; Google's Tensor Processing Unit reports an observed throughput under 23% of the peak on average and under 4% for some models [16].

A common approach to improving utilization of parallel hardware, under stochastic query load, is to leverage multi-tenancy. By sharing a GPU across multiple prediction workloads we can potentially leverage workload level parallelism and achieve statistical multiplexing. However, leveraging multi-tenancy on a GPU remains an open research problem. First, its runtime performance must be *predictable*, which, as we show in Section 3.2, is not always true. Second, it must be *resource-efficient*. To address this, we explore a number of techniques for sharing a GPU among a set of execution kernels, each with their drawbacks. Third, a measure of *performance-isolation* is needed, which is typically achieved through fair resource allocation.

Current approaches for sharing a GPU for DNN inference either multiplex the GPU across space or across time. Section 3 evaluates current approaches against the three criteria established above. We argue that only multiplexing across space or time leads to a compromise on one of these criteria – instead, we propose scheduling across space and time for GPU inference in Section 4. By packing multiple execution kernels across disjoint DNN graphs with dynamic query batching, we show potential for a multi-tenancy solution that is resource-efficient (3.2x speedup over state-of-the-art) while providing isolation and predictability.

2 Application model: A managed cloud inference service

Consider a cloud-based managed service for deploying ML models for online inference, similar to Amazon AWS SageMaker [1] or Google Cloud ML Engine [3]. Users may develop and upload their trained machine learning models to this service. The service then deploys the model onto one-or-more

replicas, which each may use CPUs and GPUs. The service and the users agree on some Service Level Objective (SLO), such as a measure of tail-latency for model inference.

We simplify this model in order to isolate interference effects due to multi-tenant execution. First, all models running on a single GPU are restricted to the same architecture (but different weights). This separates the impact of heterogeneous model architectures from multi-tenancy. Second, request queues are always saturated, thereby isolating model service latency from request queuing latency. It is worth noting that addressing both model heterogeneity and queuing latency is a key focus of future work.

3 Investigating limitations of space-only and time-only multiplexing

Using this simplified model of a managed cloud inference service, we outline three leading approaches to model inference today:

1. *Exclusive access*: Each model has an exclusive GPU. Amazon AWS SageMaker, Google Cloud ML Engine, Clipper [10] and TensorFlow Serving [19] all utilize this approach. In this approach, inference is done in batches. When the network is performing forward propagation, new queries must wait in a queue until one forward pass is completed.
2. *Time Multiplexing*: An on-device scheduler enables interleaved execution of multiple CUDA contexts at once. This approach is common when multiple processes run concurrently using the same GPU. This approach relies on the kernel to time-multiplex processes and GPU to swap contexts when different processes compete for the same resource.
3. *Spatial Multiplexing*: Kernel execution can overlap by utilizing NVIDIA Hyper-Q [6]. CUDA Streams and NVIDIA Multi Process Service (MPS) [5] utilize multiple hardware queues to enable spatial sharing of the GPU. CUDA Streams is used by ModelBatch [18] and NVIDIA TensorRT [7]. AMD's MxGPU (SR-IOV) [2] is another approach for spatial multiplexing, not considered in this work. In this paper, we consider two kinds of spatial multiplexing:
   (a) *Implicit Spatial Multiplexing with MPS*: NVIDIA MPS allows multiple processes to run on the device at the same time by allocating them different cuda streams.
   (b) *Explicit Spatial Multiplexing with CUDA Streams*: With this method we directly interact with multiple CUDA streams inside a single processes.

We examine competitive solutions for each of these model inference approaches to identify if they satisfy our chosen system criteria, namely: *latency predictability*, *resource-efficiency*, and *performance isolation*.

3.1 Experimental setup

We evaluate these three virtualization methods on two image classification neural network architectures: MobileNet V2 [22] and ResNet-50 [12]. These two models are popular choices for low-compute and high-accuracy classification applications respectively.

1. *Exclusive access* is tested with a single model executing batched queries on a private GPU. Although we cannot use this approach with multiple models on a single GPU, this test represents a single-tenant lower bound on latency and an ideal best-case for performance.
2. *Time multiplexing* is tested by running each model in a separate CUDA context and utilizing a software scheduler to interleave execution. This provides memory safety and some basic level of isolation between tenants.
3. *Spatial multiplexing* is tested by using the NVIDIA MPS server to partition model queries for different models across a pool of CUDA streams.

All experiments use p3.2xlarge or p3.8xlarge instances on Amazon AWS. These instances have direct access to NVIDIA V100 datacenter-class GPUs with up to 14 TFLOP/s of single-precision floating-point throughput. We do not test Tensor Core sharing in our experiments.

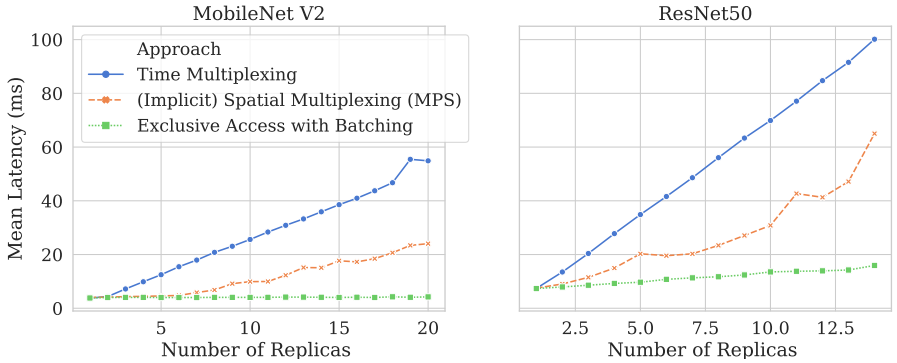

Figure 3: **Both time and spatial multiplexing do not meet the performance of exclusive access; however, spatial multiplexing is able to deliver better inference latency than time multiplexing.** We compare three approaches to GPU multi-tenancy. Exclusive access (modeled by batching a single model) provides fast and predictable latencies at a high cost. Time multiplexing dramatically increases inference latency as sharing increases. Spatial multiplexing through NVIDIA MPS better manages latency by sharing resources.

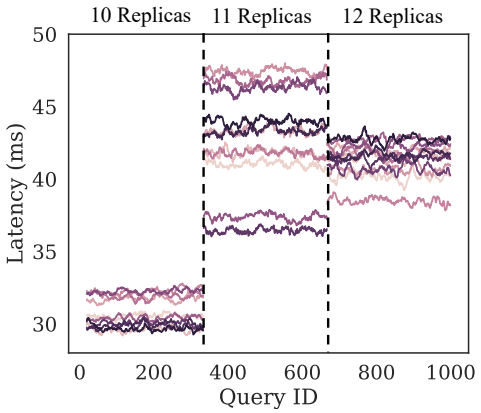

Figure 4: **Implicit spatial multiplexing (with MPS) has unpredictable latency when different number of processes are running concurrently.** As we add replicas to a GPU running 10 multi-tenant models, we observe unpredictability, which we suspect is caused by the on-device scheduler.

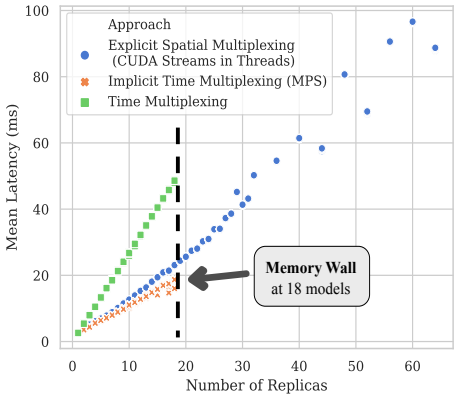

Figure 5: **Both time and implicit spatial multiplexing are bounded by memory. Explicit spatial multiplexing is not.** In this experiment, most approaches hit a 16 GB memory wall at 18 replicas, at which point GPU memory was exhausted; however, explicit spatial multiplexing (CUDA Streams on different threads), was able to scale up to at least 60 ResNet-50.

3.2 Preliminary results

We report results from our benchmark in Figure 3. For both MobileNet V2 (compute-optimized model) and ResNet-50 (high-accuracy model), we tested batched exclusive access, time multiplexing, and spatial multiplexing. Batched exclusive access devotes the entire GPU to a single model, unlike the other two approaches. This is an extremely aggressive baseline and represents the ideal performance a model would achieve if it were the only tenant.

Overall, time-only multiplexing suffers a 4.6x slowdown compared to exclusive access (i.e. $L_{\text{exclusive}}/L_{\text{time}}$) while space-only multiplexing only endures a 2.2x slowdown.

Ultimately, no single solution wins on all criteria we established in Section 1.

Exclusive access, as discussed, allows for high-throughput, low-latency, isolation, and predictability, but is extremely expensive, since it does not share the GPU at all. As we note in Figure 2, this increase in performance comes at a tradeoff - namely that online inference workloads must endure low GPU utilization in order to meet these tight latency SLOs. We believe this is a great model for users who have high enough request rates during inference to achieve good utilization on a GPU and are able to batch requests to arrive simultaneously into one forward pass of the network.

Time multiplexing (CUDA context switching) can actually accomplish multi-tenancy with good isolation and predictability, but at the cost of degraded throughput and high latencies. The main drawback to this approach is that it cannot take advantage of parallel execution of the kernels, since the GPU only allows one running CUDA context at a time. This approach instead interleaves processes resulting in slightly improved resource-efficiency; although it still suffers from poor utilization during each schedule quantum, explaining the linear-slowdown as the number of replicas grows. Poor latency scalability makes time multiplexing alone an inadequate solution for interactive inference query serving.

Spatial multiplexing (Hyper-Q) does improve on poor utilization and achieves much better resource-efficiency. However, we find there is poor predictability and isolation in this setup. It seems the spatial multiplexing approach is extremely sensitive to the choice of the number of tenants. Each tenant appears to have fairly consistent behaviour once the model runs. But, across multiple model tenants, there is up to a 25% latency gap between the fastest model on a GPU and the slowest straggler model as seen in Figure 4. Furthermore, the unpredictability and discrepancy between latencies across different processes is exacerbated when an odd number of processes runs concurrently with MPS enabled.

4 A new hope? Dynamic space-time scheduling

From our evaluation of current approaches, we find that neither time-only multiplexing nor space-only multiplexing can meet all three performance criteria: good resource efficiency, isolation and predictability. GPU single-tenancy leads to poor utilization and high costs (Figure 2). Figure 3 and Figure 5 demonstrate inherent scalability limitations to common space-only and time-only multiplexing strategies. Figure 4 details unpredictable latency as tenants are added to a GPU. Figure 5 shows the only way to schedule hundreds of models on GPU is to use single process utilizing one CUDA stream per thread; since we are micro-managing the inference by dispatching kernels to different streams, there is an opportunity for more fine-grained scheduling control over the streams to optimize latency and throughput.

In light of these limitations, we propose a promising new approach we call *dynamic space-time scheduling*. The key idea is trade-off space and time multiplexing in order to efficiently utilize the GPU while preserving isolation and predictability.

We preserve predictability and isolation during virtualization by monitoring inference latencies per-kernel. This allows reallocating resources between tenants on-the-fly. Moreover, we notice that CUDA Stream scheduling anomalies typically only create a few stragglers, so we can simply evict degraded workers without significantly impacting total system throughput. We are further investigating this approach in ongoing work.

Our approach also dramatically improves resource-efficiency on the GPU – we observe a 7.76 overall speedup in throughput compared to time-only multiplexing and a 3.18 speedup compared to space-only multiplexing, as shown in Figure 7 (geometric mean). Space-time scheduling merges many concurrent small kernels from disjoint DNN graphs into a small set of larger super-kernels that together fill the GPU. The super-kernel avoids the scheduling penalty associated with current space-only multiplexing approaches.

As interactive inference queries arrive stochastically, we cannot easily precompute super-kernels ahead-of-time. Instead, the space-time scheduler must dynamically schedule kernels as they arrive. We are investigating the design of a more general dynamic scheduler, but we notice that overheads gradually decrease if we cache super-kernels as workloads stabilize over time.

Our goal is to optimize a batch of distinct models dynamically, in comparison to ahead-of-time DNN graph optimizers like TVM [9], Tensor Comprehensions [25], Halide [20], GLOW [21] and TensorRT [7]. These optimizers excel at single-tenant optimization though kernel-fusion and auto-

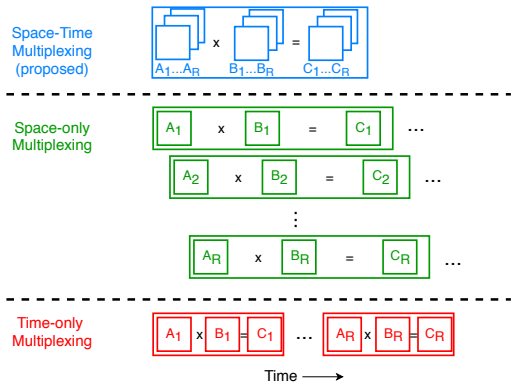

Figure 6: **Space-time scheduling reduces kernel invocations via inter-model batching.** We illustrate kernel multiplexing methods for R SGEMMs scheduled on the same GPU. Outer boxes depict a single CUDA kernel invocation.

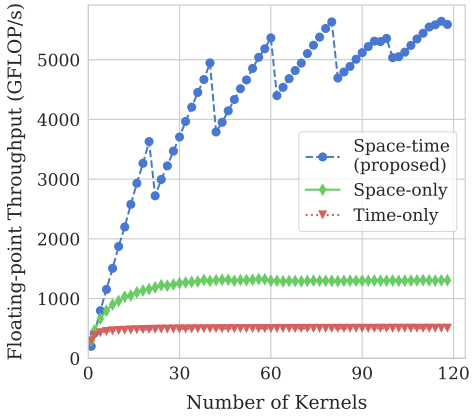

Figure 7: **Inter-model kernel batching offers ideal convolution throughput scaling.** We evaluated the matrix multiply throughputs on an intermediate ResNet-18 convolution kernel.

tuning. However, our approach focus on optimizing the performance of many disjoint graphs. Our approach extends these tools and is complementary to the this existing ecosystem.

4.1 Benchmarking dynamic space-time kernel scheduling

We evaluate the overall throughput provided by multiplexing matrix multiplication kernels by time-only, space-only, and a proposed space-time multiplexing strategy. Specifically, we examine the Single Precision floating-point General Matrix Multiply kernel (SGEMM). Matrix multiplication is often used to implement the convolution operator in neural networks, in addition to Fourier-domain, Winograd-domain, and direct kernel implementations [8].

Figure 7 demonstrates that spatial multiplexing via Hyper-Q/CUDA stream usage can improve throughput as compared to timeslicing approaches to GPU sharing. However, the average throughput is still substantially lower than the single-precision throughput offered by the V100 (Section 3.1).

Instead of maintaining separate kernel streams which the device can schedule at a fine granularity, we investigate a software-based scheduler that batches kernels across models. By batching kernels across many models into a single super-kernel, a single GPU invocation would have an opportunity to saturate all resources on the GPU for its timeslice.

To model the performance of a space-time multiplexing software scheduler, we measure the throughput of a nominal approach — collecting SGEMM problems from several models of the same architecture into a single batched matrix multiplication super-kernel. These models are have different weights and inputs, as is likely in a multi-tenancy setting. The batched super-kernel is more efficient than many smaller kernel invocations, and also better spatially multiplexes the GPU. This also allows better predictability of latency as the dynamic kernel scheduler can selectively batch kernels and determine when to execute workloads based on per-model SLOs.

Figure 7 demonstrates substantially better throughput scaling via batching than via time-only and spatial-only multiplexing approaches. Fixing the problem size $M = 256,\ N = 128,\ K = 1152$, the number of concurrently submitted SGEMM problems is scaled. This problem size represents an im2col SGEMM implementation of a representative intermediate convolution in the ResNet-18 architecture (`conv2_2`), with a 128×128 image input to the network, convolutional kernel size 3×3, and 128 input and output channels to the layer. Similar throughput improvements are observed for other intermediate layers.

This matrix multiply super-kernel is implemented in the NVIDIA cuBLAS operation `cublasSgemmBatched`. It requires all sub-kernel problem dimensions be the same. However, the MAGMA BLAS library [4] implements a variable-sized batched SGEMM that would allow for

different kernels to be batched. For all compared approaches, data is preallocated on the device as in a real-world DNN inference setting.

5 Conclusion and Future Directions

In this work, we evaluated spatial and temporal multiplexing techniques to support inference across multiple models on a single GPU. We first considered standard approaches utilized by popular DNN frameworks and GPU vendors like NVIDIA. While these techniques improve utilization, they increase latency and variability in prediction performance in benchmarks. Neither space-only nor time-only mutliplexing techniques could achieve high resource-efficiency, predictable latencies and isolation.

We observe a 5x performance gap from batch-level parallelism, suggesting substantial opportunities to improve utilization. We propose a dynamic space-and-time scheduler that addresses all three aforementioned criteria. Software-level fusion of kernel operators across multiple models and inputs presents a promising approach to online inference scheduling.

As an early evaluation of this approach, we studied roof-line performance [26] available via SGEMM fusion of all queued problems, which offers throughputs that scale well with the number of GPU tenants or model replicas. We demonstrate a $>$ 3x speedup compared to the prior state-of-the-art in online inference multitenancy. We believe this work points towards a new approach to efficient multi-tenant execution of deep neural networks through intelligent inter-model fused kernel scheduling.

Acknowledgements

We thank Koushik Sen, Eyal Sela, Zongheng Yang, Anjali Shankar and Daniel Crankshaw for their insightful feedback and edits. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[1] Amazon SageMaker on AWS. `https://aws.amazon.com/sagemaker`.

[2] AMD MxGPU: Hardware-based virtualization. `https://www.amd.com/en/graphics/workstation-virtualization-solutions`.

[3] Google Cloud Machine Learning Engine. `https://cloud.google.com/ml-engine`.

[4] Matrix algebra on gpu and multicore architectures. `https://icl.utk.edu/magma/index.html`.

[5] NVIDA multi-process service. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[6] NVIDIA Kepler GK110 whitepaper. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-architecture-whitepaper.pdf`.

[7] NVIDIA TensorRT. `https://developer.nvidia.com/tensorrt`.

[8] Why GEMM is at the Heart of Deep Learning. `https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/`.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[11] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 620–629. IEEE, 2018.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. 2018.

[15] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[16] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[18] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. Accelerating model search with model batching. 2018.

[19] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017.

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[21] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[22] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[23] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search.

[24] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[25] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[26] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.